AN APPROACH TO SOLVE SCHEDULING PROBLEMS WITH ATRRIBUTES ASSOACIATED TO THE ACTIVITIES

Francisco Ibáñez, Daniel Díaz Araya, Raymundo Forradellas
LISI – Laboratorio Integrado de Sistemas Inteligentes
Idel - Instituto de Informática - Universidad Nacional de San Juan
Cereseto y Meglioli – 5400 San Juan – Argentina
Tel: +54 261 426 47 21 - Fax: +54 261 426 51 01
{fibanez, ddiaz, kike}@iinfo.unsj.edu.ar
www.lisi.com.ar

Keywords: Scheduling, Constraint Programming, OOP, Artificial Intelligence

Abstract

This paper shows an algorithm to deal with a scheduling problem that requires to deal with conditions related to the selections of activities and preferences for the selection of resources.

The scheduling problem could be initially thought as a Flow Shop Scheduling (a particular case of a Job Shop Scheduling) [French,82]. The main difference respect to a Flow Shop Scheduling is that in our problem each activity does not require a (specified) machine but rather a member of a (specified) group of machines. Roughly speaking, we could say that discrete resources have to be considered instead of unary resources [OPL,99] [I,Sch-R,99], (many machines with identical characteristics to perform each process instead of just one machine for each process). Nevertheless, the output must specify the particular machine used by each activity and therefore the machines with identical characteristics have to be distinguished and as a result, sets of alternative resources have to be used instead of discrete resources [I,Sch-R,99].

The scheduling problem to be solved consist of *n* jobs to be performed using *m* groups of machines.

Each job is described as a list of *m* activities of given processing times, to be executed in that order.

Each activity uses a member of the associated group of machines.

For each group of machines, there is a list of attributes.

Each activity that uses a member of the corresponding group has values defined for each one of the attributes.

Besides, there is a list of conditions associated to each attribute. Depending on the values of the attributes associated to the activities, the conditions will be fulfilled or unfulfilled. Formally, the conditions are defined as a function of the values of the attributes of the involved activities.

In addition, the convenience of selecting a particular machine depends on the values of the attributes of the activity to be scheduled.

The problem is to assign each activity to a starting time and to a specific machine, trying to verify the conditions aforementioned on one hand, and selecting machines according to the values of the attributes of the activities on the other hand.
Finally, conditions (related to the selection of activities) and conveniences (for selecting machines) have associated weighs.

This work shows an algorithm to select activities and machines taking into account these weighs.

1. Introduction

Sets of alternative resources and durable resources [I,Sch-R,99] have been straightaway used in the implementation of our scheduling problem.

Although ILOG tools provide support for selecting activities, our problem requires dealing with conditions related to the selection of activities and preferences for the selection of machines, for which direct support was not found.

The scheduling problem to be solved consist of n jobs $J_1, ..., J_n$ to be performed using m group of machines $G_1,...,G_m$.

Each job J_i *(1£i£n)* is described as a list of m activities $A_{i1}, ..., A_{im}$ of given processing times, to be executed in that order.

Each activity A_{ij} *(1£i£n, 1£j£m)* uses a member of the group G_j (any member of the group G_j is able to perform the process j).

For each group of machines G_j, there is a list of attributes $AttList_j$.

Each activity that uses a member of the group G_j has values defined for each attribute of the list $AttList_j$.

Besides, there is a list of conditions associated to each attribute.
Formally, let us consider the following assumptions.
$Att_{j1},..., Att_{jk}$ is the list of attributes $AttList_j$ (there are k attributes associated to the process j) and $Cond_{j1},..., Cond_{jk}$, is the list $CondList_j$ of the corresponding conditions associated to the group G_j.
M is a member of G_j.
The activity A' has been scheduled after activity A on machine M. The values corresponding to the attributes $Att_{j1},..., Att_{jk}$ for A are $v_{j1}, ..., v_{jk}$, and those corresponding to A' are $v_{j1}', ..., v_{jk}'$.
The conditions that should be (partially at least) verified are

$$Cond_{j1}(v_{j1}, v_{j1}'), ..., Cond_{jk}(v_{jk}, v_{jk}').$$

Besides, the convenience of selecting a particular machine depends on the values of the attributes of the activity to be scheduled.

In other words, if an activity A_{ij} uses a member of the group G_j, the algorithm must try to assign A_{ij} to a specific machine (a member of G_j among those that are available).

The order in which the available machines will be tried is a function of the values of the attributes of A_{ij}.

The problem is to assign each activity to a starting time and to a specific machine, trying to verify the conditions aforementioned on one hand, and selecting machines according to the values of the attributes of the activities on the other hand.
Let us now consider the simple case in which we have machine *M*, and three conditions $cond_1$, $cond_2$ and $cond_3$ that should be studied. We assume furthermore that $cond_1$ is more important than , $cond_2$, and , $cond_2$ is more important than $cond_3$. Let act_1 and act_2 be two activities (requiring *M*) that can be scheduled at a certain time. In addition we assume that act_1 complies with $cond_1$ but it does not complies with the remaining conditions, while act_2 does not complies with $cond_1$ but fulfil the remaining. Which activity should be chosen to schedule first?

On the other hand, there are criteria to choose a particular machine.

Since the conditions are competitive, our approach allows to assign weights that represent the different degrees of importance.

The algorithm presented here, repeatedly finds an activity to be scheduled taking into account the mentioned conditions, and chooses a machine using criteria that depends on the values of the attributes of the activity. When the conditions and the criteria for choosing machines compete, the weights are regarded.

The problem has been modeled, using alternative resource sets [I,Sch-R,99] (alternative resources for [OPL,99]). From now on alternative resource sets will be referred as AltResSets.

An AltResSet is a compound resource that contains two or more equivalent resources, called alternative resources , to which activities can be assigned.
An AltResSet is defined for each process. Each AltResSet represents a set of machines such as $\{M_1, ...,M_k\}$ and contains *k* alternative resources that represent the machines $M_1, ...,M_k$.

The paper is organized as follows.

First, an algorithm to select activities and alternative resources is shown.

Thereafter, the main inherent details of the implementation, and the obtained results are mentioned.

Finally, the conclusions and the future work lines are presented.

2. Preliminary definitions

This section includes the definition and description of the functions used in the algorithm for selecting Activities and Alternative Resources.

It has been assumed that each activity requires only one AltResSet.

Let *AltResSets*, *AltResources*, and *Conditions* represent: all the AltResSets, all the alternative resources, and all the conditions, respectively. Below we included the functions involved in the algorithm.

2.1. Definition of Functions

StartMin: Activities ® N_0

N_0 represents the set of the non negative integer numbers (natural numbers with zero included) and *P(S)* denotes the set of parts of the set *S*.

AltResSet: Activities ® AltResSets

Verify: Activities X AltResources X Conditions ® {0,1}

Conds: AltResSets ® P(Conditions)

Possible: Activities X AltResources ® {0,1}

Weight: Conditions ® N

N represents the set of positive integer numbers.

AltRes: AltResSets ® P(AltResources).

If *altResSet1*, *altResSet2 Î AltResSets*, and *altResSet1 ¹ altResSet2*, we impose the following condition:

Conds(altResSet1) Ç Conds(altResSet1) = Æ

and

AltRes(altResSet1) Ç AltRes (altResSet1) = Æ

The justification of this restriction for the case of the function *AltRes* is based on the fact that each alternative resource must be associated with just one AltResSet.

In the case of the function *Conds*, it is not strictly necessary to respect the previously mentioned restriction (since it could have a condition applicable to two different AltResSets). However, it is convenient to use two different identifiers in order to simplify the algorithm. For example, without this restriction, the function *Weight* would require an extra argument to take into account the AltResSet considered.

AltResPreference: Activities X AltResources ® N

AltConvenience: Activities X AltResources X P(Conditions) ® N_0

AltResSetConvenience: Activities X AltResSets ® N_0

ActivityConvenience: Activities ® N_0

2.2. Description of the functions

StartMin: Takes as argument an activity not scheduled, and returns the minimal possible start time.

Given an activity *act*, the mechanism used to evaluate *StartMin(act)* guarantees on one hand that it will return a value (let us say *startMin*) and on the other hand that it will be possible to schedule the activity *act* at the time *startMin*.

Regarding alternative resources, this implies that there are at least one alternative resource, available for the activity *act* at the time *startMin.*

AltResSet: takes as argument an activity, and returns the required AltResSet.

Verify: takes as arguments an activity *act*, an alternative resource *altRes*t, and a condition *cond*, and returns 1 if *act* verify the condition *cond* at the time *StartMin(act)* with respect to the alternative resource *altRes*t. Otherwise the function returns the value *0*.

Conds: take as argument a *AltResSet*, and returns the set of conditions associated with the argument.

Possible: takes as arguments, an activity *act*, and an alternative resource *altRes*, and returns the value *1* if it is possible to assign the alternative resource *altRes* to the activity *act* at the time *StartMin(act)*. Otherwise it returns *0*.

Weight: Takes as argument a condition and returns a value that represents the degree of importance of that condition.

AltRes: takes as argument an AltResSet and returns the set of alternative resources that are part of the AltResSet.

AltResPreference: takes as arguments, an activity and an alternative resource, and returns a non negative integer number, whose value is set according to the convenience of assigning the alternative resource to the activity.

Given,

an activity *a*,

an AltResSet *AltResSet*,

an Alternative Resource *altRest* $\in$ *AltRes(AltResSet)*,

and *conds = Conds(AltResSet)*,

the functions *AltConvenience*, *AltResSetConvenience* and *ActivityConvenience* are defined as follows:

AltConvenience(a, altRest , Conds) =

$$Possible(a, altRest) * (AltResPreference(a, altRest) + \sum_{c \in conds} Verify(a, rAlt, c) * Weight(c))$$

AltResSetConvenience(act, recDiscr) =

$$Max_{recAlt \in AltRes(recDiscr)} AltConvenience(act, recAlt, Conds(recDiscr))$$

ActivityConvenience(act) = AltResSetConvenience(act, AltResSet(act)),

3. Selecting Activities and Alternative Resources

3.1. Algorithm

1 $Min = Min_{act \in Activities} StartMin(act)$

(Get the minimum time in which it is possible to schedule an activity).

2 $MinSet = \{act \in Activities : StartMin(act) = Min\}$

(Get the set of activities with minimum start time *Min*)

Notice that $MinSet \neq \emptyset$, since $Activities \neq \emptyset$.

3 $MaxConvenience = Max_{act \in MinSet} ActivityConvenience(act)$

4 $Pairs = \{(a, altRes) : a \in MinSet, rDiscr = AltResSet(a), \quad altRes \in AltRes(rDiscr),$
$Conds = Conds(rDiscr)$,
$AltConvenience(a, altRes, Conds) = MaxConvenience \}$

(Get the set of pairs Activity-AlternativeResource that maximise the function *AltConvenience*).

5 Select an element of the set Pairs.

3.2. Features of the strategy

The elections of the values for the function *Weight* determine different strategies for choosing activities.

Let us consider the simple case of a *AltResSet* formed by just one alternative resource *r*, with conditions $cond_1, ..., cond_n$. Let act_1 and act_2 be two activities candidates to be

scheduled at a given time using the alternative resource *r*. We also suppose that act_1 verify $cond_1$ and it does not verify the remaining conditions, while act_2 does not verify $cond_1$ and verify the remaining ones..

Lets us consider

$Conds(r) = \{cond_1, ..., cond_n\}$,

If we choose

$Weight(cond_i) = c$ ($1 \leq i \leq n$ and c is a constant)

we will obtain a strategy that chooses an activity that verify as many conditions as possible. For the previous particular case, the activity act_1 will be chosen.

If we choose instead

$Weight(cond_i) = 2^{i-1}$ $(1 \leq i \leq n)$

we will obtain a strategy that will consider the condition $cond_i$ more important than all the other conditions $cond_j$ (for $j>i$). For the previous particular case, the activity act_2 will be chosen.

For the case of two AltResSets, it would be possible to assign weights to one of the AltResSets, getting the first strategy and to assign other weights to the other AltResSets, getting the second strategy. For more AltResSets, it is possible to assign weights to each AltResSet, as a function of the degree of importance of the conditions associated with each AltResSets.

On the other hand, different values for the function *Weight* and for the function *AltResPreference* determine different strategies to choose an activity, when alternative resources have to be considered.

Let us consider the simple case of just one AltResSet *r*.

Let us assume that *r* is formed by *m* alternative resources: $altRes_1, ..., altRes_m$.

Let us consider

$Conds(r) = \{cond_1, ..., cond_n\}$

If we choose

$Weight(cond_i) = m*2^{i-1}$ and

$AltResPreference(act, altRes_j) = j$ $(1 \leq j \leq m)$,

we will obtain a strategy that considers more important the conditions associated with *r*, than the preferences of the alternative resources.

In fact all the second components of the pairs *(act, altRes)* of the set *Pairs* are exactly the same.

Formally,

(1) If $(act, altRes) \in Pairs$ and $(act', altRes') \in Pairs \Rightarrow altRes = altRes'$.

On the other hand, the conditions that verify all the activities that belong to a pair of the set *Pairs* are exactly the same, and therefore:

(2) If *(act, altRes)* Î *Pairs* and *(act', altRes')* Î *Pairs* ⇒

Verify(act, altRes, condi) = Verify(act', altRes', condi) (1£i£n)

Consequently, it could have been possible to consider first the conditions that must verify an activity to be selected, disregarding the function *AltResPreference*, and find the (unique) alternative resource in a subsequent step.

On the contrary, if we assign to the function *AltResPreference*, values greater than those assigned to the function *Weight*, we will establish more importance to the selection of the alternative resources than to the conditions.

The most interesting issue, from the practical point of view, is that for each AltResSet, it is possible to assigned different weights (for the conditions and for the selection of the resources), generating strategies adapted for each case.

3. 3. Implementation

The application have been implemented in C++, employing routines of Ilog Scheduler [I,Sch-R,99] and Ilog Solver [I,Sol-R,99].

Because it is necessary to know the values of the attributes of each alternative resource, at any time, it was necessary to implement our own timetables, in order to include the managing of this information.

A timetable for an alternative resource can be implemented in C++ using a package of objects of a type T [Strou, 97]. Each one of the objects represents an interval of availability of the alternative resource.

The minimal necessary data to represent an interval of availability are the *start* and the *end* of the interval. In order to associate a set of attributes to each interval of availability, it is necessary to know the particular alternative resource. All the alternative resources that form part of the *AltResSet* to which they belong, share the same attributes, but different AltResSets have different attributes.

The solution adopted was to define a class *AvailableInterval* with just two data members that represent the start and the end of the interval of availability, and to define, for each *AltResSet*, a subclass, inherent from *AvailableInterval*, that includes as data members, those necessary to represent the particular attributes of *AltResSet*.

Therefore there are as many classes (derived from *AvailableInterval*) as *AltResSets* in existence.

Each alternative resource will be an instance of a package. If *altRes1* form part of the AltResSet *AltResSet1*, the type of the objects that constitute the package for *altRes1* must be the subclass defined for *AltResSet1*. *Let us call it AltResSet1Class*.

To define timetables for alternative resources that form part of different alternative resources, we use templates, and we define a class *TimeTableWithAttributes* in the following way:

template <class Generic> class TimeTableWithAttributes: public vector<Generic>

In order to generate a time table for the alternative resource *altRes1*, we created the instance *altRes1TimeTable* like that:

TimeTableWithAttributes < AltResSet1Class > altRes1TimeTable(n);

where *n* is the initial size of the package.

4. Obtained Results

The algorithm has been tested with 1,500 activities, and 6,000 constraints. The execution stops immediately after obtaining the first solution, so it is necessary to make extreme efforts to achieve an acceptable first solution. In our implementation, we do not use an objective function to minimize, but rather we provide different measures to evaluate the quality of the results. Between these measures are, the percentage of conditions that are verified, and the measures related to the due date. It is difficult to obtain an average behavior in terms of execution time or in terms of percentage of conditions verified, due to the fact that the output is strongly dependent on the particular input data.

Unfortunately the data are to big to be analyzed in this paper. However, we can comment on two relevant problems that arose from some particular input data.

On one hand, for the first implementation of the algorithm the program did not respond within an acceptable period of time, and on the other hand, in spite of a considerable increase in the percentage of conditions verified, some due dates could not be reached.

The reason for the last inconvenience is due to the following fact. Initially, the activities are ordered according to the due dates. Without conditions, the activities tend to be scheduled in an order relatively similar to the original one. When conditions are imposed, it might produce a considerable postponement of urgent jobs, due to the fact that they do not verify enough conditions.

The solution adopted to overcome these problems was to divide the set of activities into segments. Each segment considers the conditions without relationship to the previous segment and consequently, at the beginning of each segment, the scheduled activities do not necessarily fulfill enough of the conditions.

The timetables of the resources last throughout all of the execution processes, but the memory required to store the data related to the activities and to the restrictions, is released each time a segment ends. Both the main memory size required and the quantity of constraints that need to be handled in each segment, decrease proportionally to the size of the segment, while the execution time is reduced drastically, as the size of the segment decreases.

For example, for one segment of 1,500 activities, there are 6,586 constraints, the main memory required is 7,949,020 bytes and the execution time is 22.79 minutes, while for 5 segments, the corresponding values are the following: 1,572 constraints, 1,108,996 bytes, 30 seconds.

In spite of the previously mentioned advantages, the segmentation introduces a degradation of the quality of the solution, with respect to the use of the machines, owing to those activities that are scheduled in the first segments originate idle periods of time for the machines. The activities belonging to the following segments are scheduled using these periods of time, but the difference between the period of time within an activity was

scheduled (idle period of time generated in the previous segments) and the duration of the activity is wasted.

Summarizing, very big segments can lead to the violation of the due dates and segments too small, degrade the use of the machines.

In order to find a compromise between these two possibilities, the user can choose the (estimated adequate) size of the segments.

5. Conclusion and Future Lines

In this work, an algorithm for choosing activities and alternative resources has been presented. These selections are based on the evaluation of conditions that depend on the characteristics of the machines. Different strategies arise from different weight assignments.

It is possible therefore, to generate different priorities for the conditions and for the alternative resources. The former priorities will be considered to select activities while the latter will be taken into account to select the alternative resources.

Each set of weight assignments for a given AltResSet, generates a policy to select activities and alternative resources for this AltResSet.

For each AltResSet, the user has to choose a set of assignments according to the particular characteristics of the AltResSet.

Although the application of the algorithms lead to schedules that tend to fulfil the established conditions, two crucial, practical aspects arise:

- Some due dates are not attained on time
- The execution time of the program is prolonged

The segmentation partially solves these drawbacks but it is necessary to be careful when selecting the segment size for each particular case.

The algorithm presented here does not include the time of preparations of machines (called set up time). The main future work is to enhance the algorithm to deal with set up times for the machines.

6. References

[French,82] S. French, “Sequencing and Scheduling”, Ellis Hornwood Ltd, 1982.

[I,Sol-R,99] “Ilog Solver - Reference Manual Version 4.4”, Ilog, France, 1999.

[I,Sch-R,99] “Ilog Schedule- Reference Manual Version 4.4”, Ilog, France, 1999.

[OPL,99] P. Van Hentenryck The OPL Optimization Programming Language. The MIT Press. Cambridge, MA,1999.

[Strou, 97] Bjarne Stroustrup, “C++ Programming language Third edition”, Addison Wesley, 1997.